



More Performance, More Gameplay

Andreas Fredriksson
Gameplay Director, Insomniac Games

Hi, I'm Andreas

I manage the gameplay programming team at Insomniac Games

Ratchet & Clank, Sunset Overdrive, Spider-Man PS4

Today's topic: Optimization

Optimization

Need to be precise here

Optimize for CPU time?

Optimize for size? Code size? Data size?

Optimize for worst case? Best case? Average case?

Data build time? Compile time? Link time? Asset reload time?

Optimization is not magic

Q: "Tell me how you optimize something for time?"

A: "I look at the profiler and see what's taking time"

Q: "Then what do you do?"

A: "..."

A common optimization pitfall

Programmers tend to fixate at the top thing in the profiler

"How can we make physics go faster?"

"Rendering seems slow"

These are usually problems that are well solved already

What's the objective?

When optimizing for time, the objective is to get the game in frame

30 Hz = ~33 ms

60 Hz = ~16 ms

90 Hz = ~10 ms

Finding the right problem

No one cares where we get the milliseconds we need

As long as the game is in frame

Pick off easy wins first

Payoff is much quicker than attacking the big, complicated stuff

E.g. Rendering/Physics as a whole probably is already pretty optimized

Unlikely we can move the needle here quickly without a LOT of effort

Can probably knock out 3-4 small gameplay things in a fraction of the time

A story about physics..

Game: Battlefield Bad Company 2 (X360/PS3)

Assumed problem: "Physics raycasts are too slow!"

Wrong fix: Sit down and optimize physics system.

Real problem: There were multiple 4 km long raycasts that were
1) not required and
2) not used

Real fix: Remove the 4k raycasts (designer task)

A story about shadows..

Game: Sunset Overdrive (XBox One)

Assumed problem: "Shadows are too slow!"

Wrong fix: Sit down and optimize shadow map rendering

Real problem: A whole copy of the scene had been stored underground for a cinematic

Real fix: Remove this copy of the scene.

A story about alpha blending..

Game: Sunset Overdrive (Xbox One)

Problem: "Rendering of the alpha layer is too slow!"

Wrong fix: Try to generally speed up the whole VFX/post pipeline

Real problem: A full screen quad with alpha zero was always drawn due to an optional gameplay effect

Real fix: Disable gameplay system when not in use

Smart optimization

Understand the problem

Understand the data

Understand the algorithm

Understand the latency requirements

Remove waste

Use the hardware to full effect

Understand the problem

Is it the right problem to solve?

What is slow?

Can you form a theory as to why?

What can you disable, change or hack to prove or disprove the theory?

What data can you collect about the cause?

Measure

Measure baseline before you start optimizing

Find representative best, worst and average cases

Use more than one way of measuring

Sampling profilers, basic wall clock time, specialized tools

Set a reasonable goal

Continually repeat measurements to track progress

Attacking frame spikes

Intermittent frame spikes may seem like an insurmountable problem

Step 1: Gather data about what triggers the spike

Step 2: Write scaffolding that forces the spike to occur every frame

Example: Spawn 100 enemy AI every frame

Example: Force all bots to re-path their navigation every frame

Now we can use standard tools (sampling profilers etc) to work

Understand the data

Instrument systems and features to generate any data you need

Use anything available

Printf, debugger, logging to a file, debug rendering

What are the outliers?

How often does the data change?

What is the expected use of the system or feature?

How does your collected data line up with this?

Understand the algorithm

How does the time change with varying inputs?

What general class of complexity is it?

- $O(1)$? $O(\log N)$? $O(N)$? :)
- $O(N^2)$? :(
- $O(N^3)$? :((((

Understand the latency requirements

When are the results needed?

Can they be computed earlier?

Can they be computed in parallel?

Can they be (partially) precomputed?

Remove waste

What is being computed but not used?

Micro: Full matrix computed, only need position

Macro: Compute visibility for entire world, only need 10%

What is being computed over and over again?

Can you do it (partially) with a lookup table?

Can you recompute only when it changes?

Use the hardware to full effect

Assuming everything else is reasonable, how does it run on HW?

Are the data access patterns efficient?

Number of L2 misses is often a good indication

Is SIMD being used?

Can it run on multiple cores?

Can it run on GPU?

Challenge your intuition

```
float f = sqrtf(obj->radius);
```

What's slow (high latency) about this code?

Sure, there's the square root..

But the load from memory can be **10x** slower

And it can be 10x faster, as well

Things are often not what we might assume under old best practices

Moving on..

I'm assuming we've already done our research!

The data is reasonable

The systems are used in a reasonable way

The constraints are valid

There is no massive waste

So now we can talk about getting things to run well on hardware :)

AMD Jaguar

What we already know

Well rounded architecture, not too many pitfalls

Out of order execution (big change from the PS3 era)

Reasonable cache

Sounds easy!

Out of order execution (OOO)

Designed to boost "generic x86 workloads"

Not magic

OOO is at the heart of the chip's design

Important to build an intuition around OOO when optimizing

What can OOO give us, then?

Loop unrolling*

Context aware prefetching*

The ability to overlap latencies*

* Some of the time

Constant instruction fetching

Instruction fetching by speculation is part of what OOO is

Instructions will be fetched and start executing speculatively

They will request loads from memory and store reservations..

So we take not just the hit of the branch misprediction

Also have to suffer through cache effects of the "wrong" side of the branch

Branchy data structure example

```
struct Node
{
    Node *left;
    Node *right;
    BigInt bigData;
};
```


Branchy data structure example

```
void DoSomethingExpensiveToNodes(Node* n, int f)
{
    int decide = SomehowDecideChild(n, f); // high latency
    if (decide < 0) {
        DoSomethingExpensiveToNodes(n->left);
    } else if (decide > 0) {
        DoSomethingExpensiveToNodes(n->right);
    } else {
        // Do something expensive to n->bigData
    }
}
```

Misprediction central

The Battle of North Bridge

D1 Hit



L2 Hit



Memory



Are L2 misses still a thing?

An load that misses L2 will not retire for 200+ cycles

So what?

OOO can reorder around long latencies, right?

Sure, but we're always fetching new instructions

The frontend issues 2 instructions / cycle..

Retiring

All instructions *retire* (commit) in program order

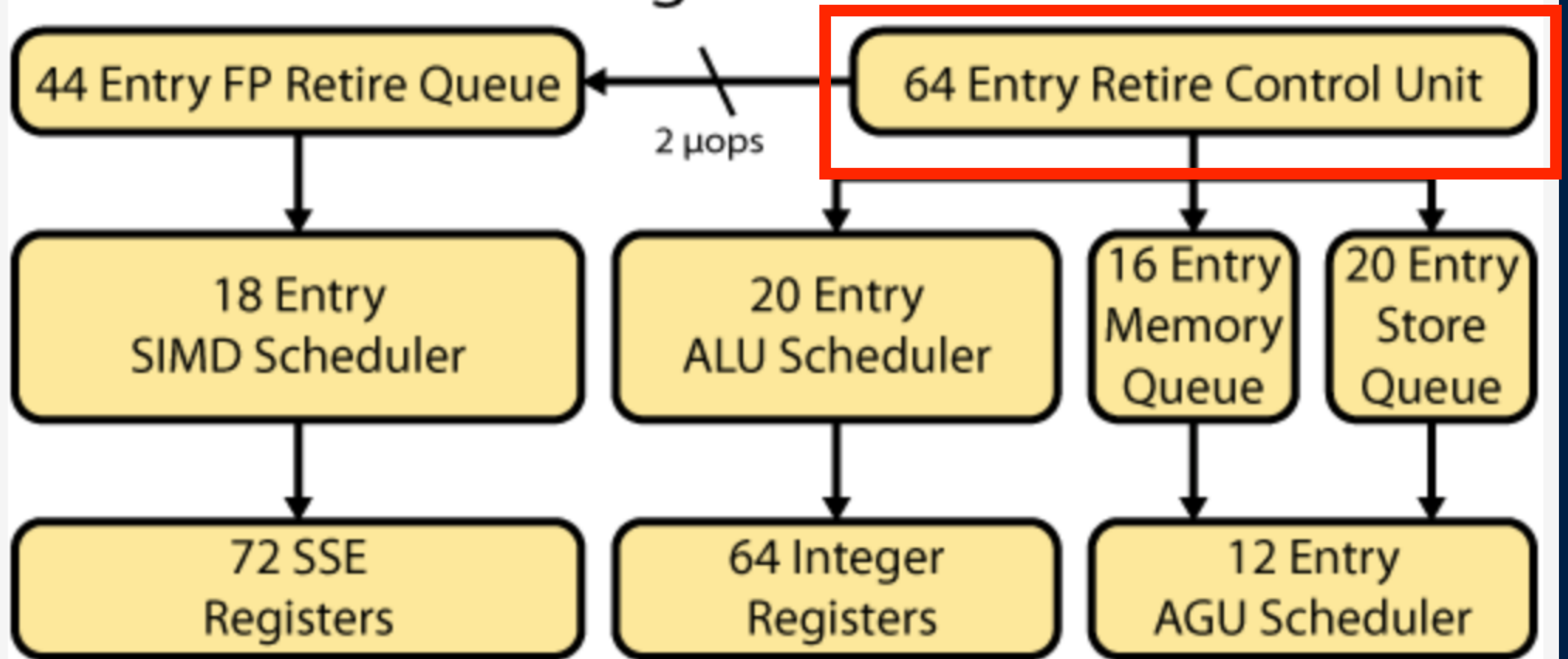
That is, their effects are visible from outside the core

Retirement happens at a max rate of 2/cycle

They can also be killed instead of retired

Due to branch mispredictions

Jaguar Core



<http://www.realworldtech.com/jaguar/4/>

Why L2 misses still matter, a lot

Assume L2 miss followed by low-latency instructions

Cache hits, simple ALU, predicted branches, etc.

Common in practice

RCU fills up in ~32 cycles

Result: ~150+ cycles wasted stalling on RCU

Only the L2 miss retiring will free up RCU space

Micro-optimizing L2 misses

Move independent instructions with long latencies to right after a load that is likely to miss

The longer the latencies, the more it softens the blow

Square roots, divides and reciprocals

And other loads..

Remember: The overlap window is small

Make the most of it!

Poor load organization

```
void MyRoutine(A* ap, B* bp)
{
    float a = ap->A;    // L2 miss
    < prep work on A >

    float b = bp->B;    // L2 miss, RCU stall risk
    < prep work on B >

    < rest of routine >
}
```


Better load organization

```
void MyRoutineAlt(A* ap, B* bp)
{
    float a = ap->A;    // L2 miss
    float b = bp->B;    // L2 miss (probably "free")

    < prep work on A >
    < prep work on B >

    < rest of routine >
}
```

L2 misses on Jaguar in practice

They're still a massive problem

We can't pretend OOO solves memory latency

Issue loads together to overlap potential misses

Hedging our bets in case more than one miss

Can "wait for" up to 8 L2 misses concurrently

Hoist loads as early as possible

Should we unroll loops?

Typically doesn't help more complicated loops

Any added latency anywhere shifts the balance

OOO is a hardware loop unroller!

The hardware will run head into "future" iterations of the loop, issuing them speculatively

Only if everything is in cache and all ops are simple will frontend dominate the loop performance

Jaguar Unrolling Guidelines

In general, turn to SIMD before you unroll scalar code

Unroll only to gather enough data to run at the full SIMD width

E.g. Unroll 32-bit fetching gather loop 4 times

Then process in 128-bit SIMD registers

Make sure the compiler is generating good code (they love unrolling)

Prefetching

Required on PPC console era chips

Sprinkle in loops and reap benefits!

x86 also offers prefetch instructions

PREFETCHT0/1/2 - Vanilla prefetches

PREFETCHNTA - Non-temporal prefetch

Use `_mm_prefetch(addr, _MM_HINT_XXX)`

So, should we use prefetches on Jaguar?

Jaguar Prefetching Guidelines

Never prefetch basic arrays

Actually hurts warm cache case with short loops

Prefetch only heavy array/pointer workloads

Need enough work to overlap the latency of the prefetch

Non-intuitive to reason about

Best to add close to gold when things are stable

Inclusive cache hierarchy fun

Inclusive cache = all D1/I1 lines must also be in L2

The L2 hears about all D1 **misses**

But the L2 hears nothing about D1 **hits**

So what if we have a routine that does nothing but **HIT** D1?

Hot D1, Cold L2

Net effect: White hot D1 data can be evicted, randomly

L2 associativity = 16 lines, so they WILL be reused frequently

Our data looks old in the pLRU order and the L2 hasn't heard about it for a while..

Result: Inner loop has to L2 miss all the way to main memory randomly to get back its really hot data

In practice not a big deal, but can definitely show up

Jaguar SIMD

SIMD is a big deal on Jaguar

Only SIMD instructions can use the full cache bandwidth

Can only read/write once to the D1 per cycle

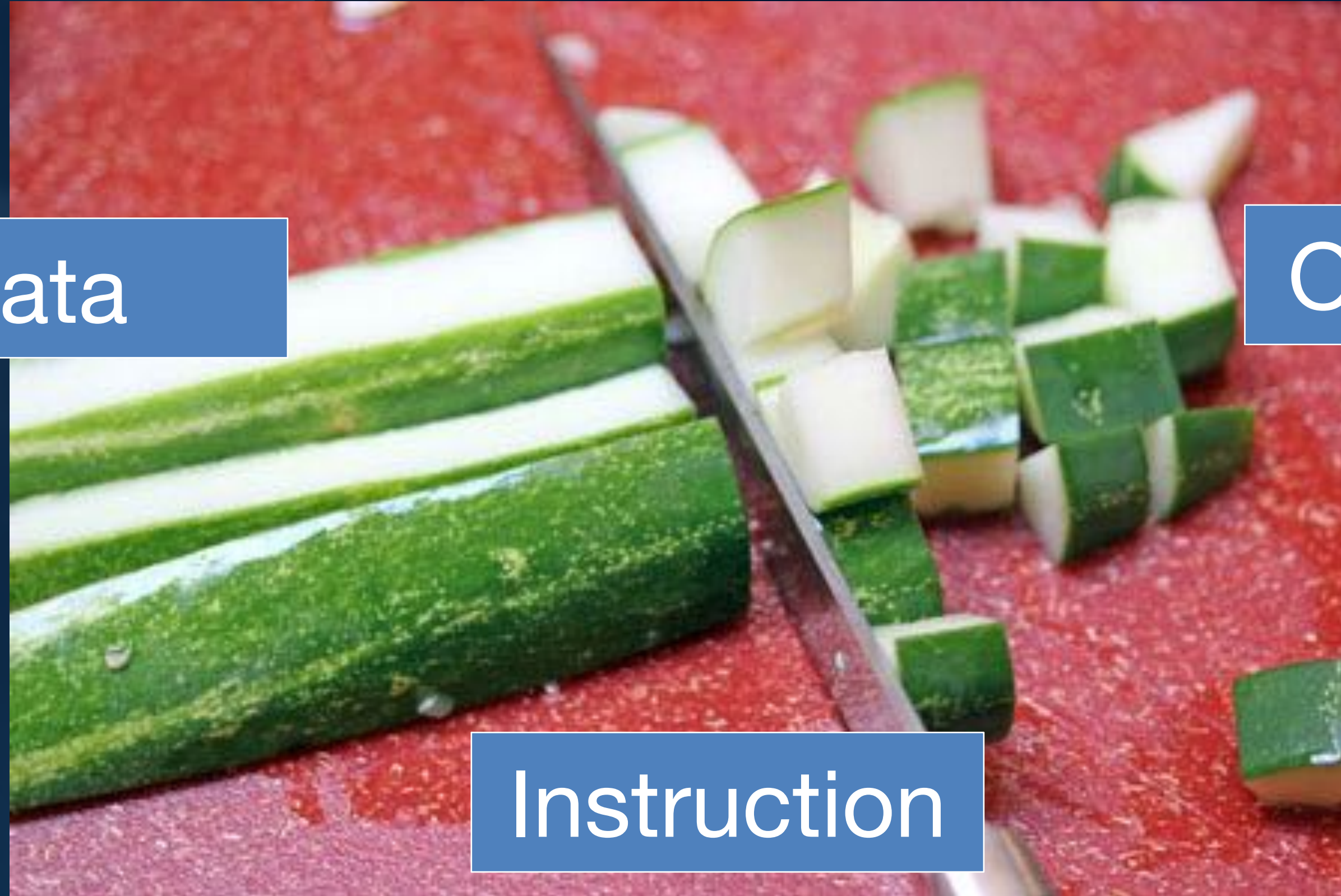
SIMD instructions can read 128 bits, scalar is limited to 64

SIMD

Input Data

Output Data

Instruction



.. It's just like dicing veggies!

Practical: Small Spatial Queries

Problem: Many "small" spatial queries in gameplay code

- Find pickups near player

- Find best cover position

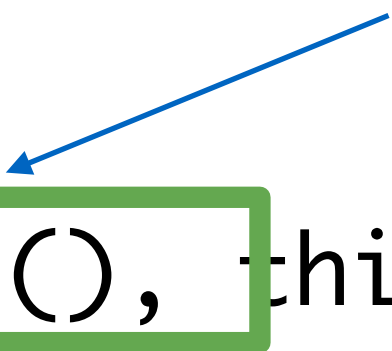
Scope: At most a few hundred things to consider

Naive brute force

```
Pickup** pickups = ...;
int count = ...;

for (int i = 0; i < count; ++i) {
    Pickup* p = pickups[i];
    float distance = Distance(p->Position(), this->Position());
    if (distance <= maxDistance) {
        // Do something with this object.
    }
}
```

'count' L2 misses



Traditional Spatial Database

Allow queries for objects in some radius

Traditional databases require expensive rebalancing

Typically stored as sphere trees or octtrees

Queries are pretty fast, but cost of updating the DB can be high

Too expensive for many dynamically moving objects

Prevents ad-hoc use for handful of objects

Our Sphere Database

Keeps only the data we need to solve the local query problem

Developed on Sunset Overdrive

Pack 4 spheres in SOA block, keep those packets in array (64 byte stride)

Parallel array of user data pointers to go along with the spheres

```
class SphereDb
{
    SphereSet4* m_Spheres;
    uintptr_t* m_UserData;
    ...
};
```

```
struct SphereSet4
{
    float x[4];
    float y[4];
    float z[4];
    float radius[4];
};
```


Sphere Database API

```
typedef int DbIndex;  
  
DbIndex Add(const BSphere& sphere, void* user_data);  
  
void Update(DbIndex index, const BSphere& sphere);  
  
void Remove(DbIndex index);  
  
int Query(void** out, int max, const BSphere& sphere);
```

Brute force query, SIMD style

Splat query bsphere into 4 SIMD lanes

For each sphere set (of 4)

Load xs, ys, zs and radii

Perform 4-wide distance test

Left-pack userdata pointers based on mask

Store userdata pointers to output buffer

Using the sphere DB

```
void* pickups[128];  
int count = pickupDb->Query(pickups, ARRAY_SIZE(pickups), sphere);  
  
for (int i = 0; i < count; ++i) {  
    Pickup* p = (Pickup*) pickups[i];  
    // Do something with this object.  
}
```

Sphere Database Summary

Cache latency bound (extremely fast)

Scales really well up to a few hundred items

Trades many L2 misses for condensed compute and cache hits

Enables us to process the interesting objects only

Cheap to update when things change (no rebalance needed)

Random writes are cheaper than random reads (store buffer)

SIMD in a game codebase

Need a good set of tricks to apply SIMD widely

Many real problems are "messy" and require special care

Comparing integers

SSE2 only has two basic integer comparisons

Compare for equality

`_mm_cmpeq_epi[8/16/32]` `PCMPEQ[BWD]`

Compare for **signed** greater-than

`_mm_cmpgt_epi[8/16/32]` `PCMPGT[BWD]`

Only greater than?

Easy to emulate other signed compares using boolean logic:

$a < b$ is $b > a$

$a \leq b$ is $\text{not}(a > b)$

$a \geq b$ is $\text{not}(b > a)$

Extra not can often be removed

E.g. for lane mask/selects, it's free if we rearrange the operands

$$\text{select}(a > b, x, y) = \text{select}(a > b, x, y)$$
$$\text{select}(a < b, x, y) = \text{select}(b > a, x, y)$$
$$\text{select}(a \leq b, x, y) = \text{select}(a > b, y, x)$$
$$\text{select}(a \geq b, x, y) = \text{select}(b > a, y, x)$$

What about unsigned compares?

Need to use biasing (see Hacker's Delight)

$$a \overset{u}{<} b = (a - 2^{31}) \overset{s}{<} (b - 2^{31})$$

$$a \overset{u}{>} b = (a - 2^{31}) \overset{s}{>} (b - 2^{31})$$

$$a \overset{s}{<} b = (a + 2^{31}) \overset{u}{<} (b + 2^{31})$$

$$a \overset{s}{>} b = (a + 2^{31}) \overset{u}{>} (b + 2^{31})$$

Computing an unsigned $A > B$ mask

```
__m128i a      = ...;
__m128i b      = ...;
__m128i bias = _mm_set1_epi32(1 << 31);
__m128i mask = _mm_cmpgt_epi32(_mm_sub_epi32(a, bias),
                               _mm_sub_epi32(b, bias));
...
```


Unsigned int -> float conversion

Trivial to write for scalar code

In fact, happens implicitly

Consider

```
uint32_t x = ...;
```

```
float f = x;
```

What's involved?

Unsigned int -> float conversion

SSE has **signed** int -> float conversion

```
__m128 _mm_cvtepi32_ps(__m128i a);
```

But there's no unsigned functionality..

Several ways we can implement this in software

The 16-bit split approach

```
__m128 Uint32_Float1(__m128i in)
{
    // Isolate low and high 16-bit parts of each 32-bit integer
    __m128i lo_int = _mm_and_si128(_mm_set1_epi32(0xffff), in);
    __m128i hi_int = _mm_srli_epi32(in, 16);

    // Convert both low and high parts to float separately.
    // These results are exact as the numbers are less than 2^16.
    __m128 loflt = _mm_cvtepi32_ps(lo_int);
    __m128 hiflt = _mm_cvtepi32_ps(hi_int);

    // Change the exponent of the high part by multiplying by 2^16
    // The result is still exact (we change the exponent only.)
    __m128 hiscl = _mm_mul_ps(hiflt, _mm_set1_ps(65536.0f));

    // Combine to final float output
    return _mm_add_ps(hiscl, loflt);
}
```

Problem: Filtering Data

Discarding data while streaming

Not a 1:1 relationship between input and output

N inputs, M outputs, $M \leq N$

Not writing multiple of SIMD register width to output!

Want to express as SIMD kernel, but how?

Scalar Filtering

```
int FilterFloats_Reference(const float input[], float output[],
                           int count, float limit)
{
    float *outputp = output;

    for (int i = 0; i < count; ++i) {
        if (input[i] >= limit)
            *outputp++ = input[i];
    }

    return (int) (outputp - output);
}
```

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {  
    __m128 val      = _mm_load_ps(input + i);  
    __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));  
  
    __m128 result   = LeftPack(mask, val);  
  
    _mm_storeu_ps(output, result);  
  
    output += _popcnt(_mm_movemask_ps(mask));  
}
```

Advance output position based on mask

Left Packing Problem (4-wide, limit=0)

	0	1	2	3	4	5	6	7
Input	1	-1	5	3	-2	7	-1	3
Mask	✓	✗	✓	✓	✗	✓	✗	✓
Left Pack	1	5	3		7	3		
Output	1	5	3	7	3			

 = Don't Care



Left Packing (SSSE3+)

`_mm_movemask_ps()` = valid lanes

Value will be in the range 0-15 for 4-wide case

Leverage indirect shuffle via `PSHUFB`

a.k.a `_mm_shuffle_epi8()`

Lookup table of 16 shuffles (4-wide case)

Each shuffle moves valid lanes to the left

Need $16 \times 16 = 256$ bytes (4 cache lines) of LUT data

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
    // Move 4 sign bits of mask to 4-bit integer value.
    int mask_signs = _mm_movemask_ps(mask);

    // Select shuffle control data
    __m128i shuf_ctrl = _mm_load_si128((const __m128i*) &shufmasks[mask_signs][0]);

    // Permute to move valid values to front of SIMD register
    __m128i packed = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

    return packed;
}
```



Shuffle YW..

Reciprocals

SSE has a fast reciprocal ($1/x$) approximation

`_mm_rcp_ps(n)`

Limitation: only 12 bits of precision

(Also exists for square root)

Can improve precision drastically using Newton-Raphson

One round is enough to give almost perfect result

Reciprocals with Newton's method

```
__m128 ComputeRecipNR(__m128 denom)
{
    // x0 = our initial "guess", with 12 bits of precision
    __m128 x0 = _mm_rcp_ps(denom);

    // x1 = refined reciprocal after one round of NR
    // Uses the formula:  $x1 = x0 * (2 - denom * x0)$ 
    __m128 x1 = _mm_mul_ps(x0, _mm_sub_ps(_mm_set1_ps(2.0f), _mm_mul_ps(denom, x0)));

    return x1;
}
```

Conclusion

Challenge your intuition

Find the appropriate problem to optimize

Remove waste

For Jaguar performance, use arrays and SIMD

Q & A

Email: afredriksson <at> insomniacgames.com

Twitter: @deplinenoise

See my GDC 2015 & 2016 talks for more stuff on SIMD and Jaguar!



Bonus Slides

Dealing with float data

Float format is easy to pick apart

And it's good to know for debugging, anyway

Knowing the bits that make up a float means we can

Deal with special floats

Handle outliers and error cases with confidence

Floats

Value Class	Sign (31)	Exponent (30:23)	Mantissa (22:0)
Positive Zero	0	0	0
Negative Zero	1	0	0
Positive Normalized	0	1..254	$\neq 0$
Negative Normalized	1	1..254	$\neq 0$
Positive Denormal	0	0	$\neq 0$
Negative Denormal	1	0	$\neq 0$
Positive Infinity	0	255	0
Negative Infinity	1	255	0
Quiet NaN	Ignored	255	$\neq 0$, MSB=0
Signaling NaN	Ignored	255	$\neq 0$, MSB=1

Dealing with NaN (Not a Number)

Usually, take one of these three approaches:

1. Allow NaNs to be generated but be careful not to use them
2. Mask away inputs that would generate NaNs before an operation
3. Check after the fact if a NaN was generated

Masking NaNs before an operation

```
__m128 input = ...;

// Generate all ones if x > 0
__m128 mask = _mm_cmpgt_ps(input, _mm_setzero_ps());

// Mask away negative inputs, replace them with zero
input = _mm_and_ps(input, mask);

// Compute square root with no negative numbers remaining
__m128 out = _mm_sqrt_ps(input);

...
```


Masking NaNs after an operation

```
__m128 input = ...;

// Generate all ones if x > 0
__m128 mask = _mm_cmpgt_ps(input, _mm_setzero_ps());

// Compute square root - this can generate NaNs based on the input data
__m128 b     = _mm_sqrt_ps(input);

// Select away NaNs based on the mask, substituting zero implicitly
__m128 out   = _mm_and_ps(mask, b);

...
```

Checking for NaNs

Take advantage of the fact that NaNs don't compare equal to anything

Even themselves!

```
NaNMask = _mm_cmpneq_ps(vals, vals)
```


Magic integer tricks

2^{23} (8,388,608) is a special float

S = 0, E = 10010110, M=000000000000000000000000

It's special because it has no fractional precision at all

Looks like 0x4b000000 in memory

What are magic ints good for?

Conversions between int and float!

Basic idea: Bitwise OR 0x4b000000 with a small integer (up to 23 bits)

Treat as float, and subtract 2^{23}

Result: That small integer converted to float

Result = `BitsAsFloat(0x4b000000|value)` - 8388608.0f

Equivalent to the cast *(float) value* for positive values $< 2^{23}$

What are magic ints good for?

Other exponents are useful as well

Use a bigger exponent to get scaling as well

0x4b000000 - x1

0x4c000000 - x4

0x4d000000 - x16

Nice way to get a free multiply into int->float conversion

The deluxe "magic int" approach

```
__m128 Uint32_Float_Magic2(__m128i in)
{
    SSE_CONSTANT_4(magic_scaled16, float, 549755813888.0f);           // 2^(16 + 23)
    SSE_CONSTANT_4(magic_unscaled, float, 8388608.0f);                // 2^23
    SSE_CONSTANT_4(magic_both, float, 549755813888.0f + 8388608.0f); // 2^23 + 2^(16 + 23)
    SSE_CONSTANT_4(lo_mask, uint32_t, 0xffff);

    __m128i lo_int = _mm_and_si128(lo_mask, in);
    __m128i hi_int = _mm_srli_epi32(in, 16);
    __m128  loflt = _mm_or_ps(_mm_castsi128_ps(lo_int), magic_unscaled);
    __m128  hiflt = _mm_or_ps(_mm_castsi128_ps(hi_int), magic_scaled16);
    __m128  hiscl = _mm_sub_ps(hiflt, magic_both);
    return   _mm_add_ps(hiscl, loflt);
}
```